This exercise is done by Zhi Mai.

1） Suppose σ1 = [t1, t2, …, tk], σ2 = [r1, r2, …, rk] are permutations of k-set. σ1 > σ2 if ti > ri, i = min{ 1《 i 《 k | ri != ti }. If the elements of σ1 and σ2 are the same, we compare their colors.

Since the number of colors is 2, we can define color1 to be 1, color2 to be 0. Then the color of each element in a permutation can be mapped to {0, 1}. So, the whole color of a permutation can be represented as a binary string. For example, the color of 1432 is 1011 where blue represents 0 and red represents 1. In other words, an element of 2-color-k-permutation is { [a1,…,ak], b1b2…bk}, bi = 0 or 1.

Then if we compare the color of σ1 and σ2, we just compare the binary number of them and see which is bigger.

2) i) For a rank, since there are 2^k colors for each permutation, we need take the remainder r and the quotient q of rank \ 2^k. The number q represents the rank of permutation, and r represents the rank of color.

To find the permutation corresponds to r, we can fix elements from left to right. Notice that each element of k-set appears only once, if ith element of a permutation is fixed, the number of possible remaining permutations is (k – i)!. When we fix the first element t1, we take the quotient r1 of r \ (k – 1)! and round r1 up (since ri 》1). Then we look for the 2nd element which is in range (t1\*(k-1)!+1, (t1+1)\*(k-1)!-1). So, to fix 2nd element, we need to take remainder r2 of r1 \ (k – 1)!. Also, ith element can’t coincide with previous all element. So, for 2nd element, if it coincides with previous all elements, we need to add it with 1 until it’s a new element. Keep this process until processing all elements. In particular, if ri = 0, it means the remaining part is the biggest permutation of current remaining elements. In other words, the remaining part is a decreasing sequence.

Finally, turn the rank of color into binary number.

The pseudo code is as follows:

Class cp: {

numberOfColor = 2;

size = k;

int permutation[size];

int color[size] }

Unrank(n):

Rank\_permutation <- n \ 2^k (round up); Rank\_color <- n % 2^k;

cp.color = encoder(cp, 2^(Rank\_color) – 1);

list = [1,..,k]

for i <- 1 to k - 1 do:

if (Rank\_permutation == 0):

for j <- i to k do:

permutation[j] = list[len(list) – j – i];

break;

position = Rank\_permutation \ (k – i)! (round up);

cp.permutation[i] = list[position];

delete list[position];

Rank\_permutation = Rank\_permutation % (k – i)!;

cp.permutation[k] = list[1];

return cp;

encoder(cp, n):

remainder <- n;

for i <- k to 1 do:

cp.color[i] = remainder % 2;

remainder = remainder \ 2 (round down);

return cp.color;

ii) To find the rank of a color-permutation(cp), we need to find the ranks of color and permutation and add 1 (since rank = #{color-permutation precede cp} + 1). For ith element, we need to find all natural numbers smaller than ith element. As previous saying, ith element can’t coincide with previous all elements and the number of possible permutations is (k – i)!. So, the number of possible permutations corresponding to ith element is #{m | m not coincides with previous all element and m < ith element} \* (k – i). And we add the numbers from 1 to k, that is, ∑#{m | m not coincides with previous all element and m < ith element} \* (k – i)!, 1《i《k. And we turn the binary string of color into integer. Since for each permutation there are 2^k colors, the rank is 2^k \* ∑#{m | m not coincides with previous all element and m < ith element} \* (k – i)! + decoder(cp.color) + 1.

The pseudo code is as follows:

Rank(cp):

Rank1 = 0;

Set = {};

for i <- 1 to k do:

for j <- I to cp.permutation[i] do:

if (j not in set)

Rank1 += (k – i)!;

Set.add(i);

Rank2 = decoder(cp);

return Rank1 \* 2^k + Rank2 + 1;

decoder(cp):

Rank = 0;

for i <- 1 to k do:

if (cp.color[i] == 1):

Rank += 2^(i – 1);

return Rank;

iii) To find the successor of a color-permutation(cp), we can first iterate color then iterate permutation. The method of finding successor of permutation is same as in the course. To find successor of binary number, we just invert all the bits from right to left until reaching a bit which is 0. Moreover, the range of binary number is [0, 2^k – 1]. So, if the binary string is 11…1, its successor is 00…0 and we need to find successor of permutation. Otherwise, we just find the successor of binary string.

The pseudo code is as follows:

Successor(cp):

Carry = 0;

if (cp.color[k] == 0):

cp.color[k] = 1;

else:

for i <- 1 to k do:

if (cp.color[i] == 1)

cp.color[i] = 0;

if (i == k):

carry = 1;

else:

cp.color[i] = 1;

break;

if (carry == 1):

l = 0;

for i <- k to 1 do:

if (cp.permutation[i – 1] < cp.permutation[i]):

l = i;

x = k;

while(cp.permutation[i - 1] > cp.permutation[x] and x >= 0):

x -= 1;

if (x == 0):

break;

swap(cp.permutation[i - 1], cp.permutation[x]);

for j <- l to l + (k - l) \ 2 (round up) do:

swap(cp.permutation[j], cp.permutation[k – (j – l)]);

return cp;

iv) So, for a k-permutation, it has 2^k different possibilities of colors. And for a k-set, it has k! permutations. Then for a k-set it has 2^k\*k! 2-color-permutations. So, we just need to use successor function 2^k \* k! times, start from {[1,2,…,k], 00…0}.

The pseudo code is as follows:

List(n):

cp.size = n;

cp.permutation = int[n];

cp.color = int[n];

for i <- 1 to k do:

cp.permutation[i] = i;

cp.color[i] = 0;

for i <- 1 to 2^k \* k! do:

print(cp);

successor(cp);

3) Outputs of functions:





